ECE 6100: Lab 4 Part F

Utility Based Cache Partitioning

Requirement:

To develop any partitioning scheme which dynamically computes the cache partition for shared caches in a multicore system.

Motivation:

Static Way Partition schemes do not consider the workload type and dynamic nature of the workload. So say, an system might want to allocate more cache ways to Application 1 during a part of its whole operation which more cache ways to Application 2 during another part as the nature of the two applications might be sure that both might be intensively accessing memory at the same time. A static way partition becomes a bottleneck in such situations.

Utility Based Cache Partitioning

Reference: https://ieeexplore.ieee.org/document/4041865

The idea behind this scheme is to compute partitioning based on the logic of utility, which is defined at the level of way granularity. This basically computes the possible increase in miss rate if the way between different cores is changed. Utility is defined as:

$$U_a^b = miss_a - miss_b$$

where a < b. To achieve this, additional structures are added to monitor the data access pattern, such as a Utility monitor (UMON) and partitioning algorithm computation engine.

Now, for utility monitoring, we add new structures to each way of the set, which is a hit counter. It tracks the number of hits for each way. Now, for utility computation, these are order from MRU to LRU. Then the possible change in miss rate can be predicted for each decrement in number of ways for this core, going from the LRU toward the MRU. This is done for both cores in our case. Using this, we compute the total utility for each possible value of way partitioning, using:

$$U_{tot}(a) = UA_1^i - UB_1^{16-i}$$

where A and B are the two applications running, 16 is the total number of ways. Our target is to find the partitioning value to maximize the above function which maximizes the utility. This can be implemented in various ways, such as local UMON, global UMON (through Auxiliary Tag Directory, ATD), spaced way prediction, etc. I have chosen to implement the local UMON approach.

Code Implementation:

Data Structure Updates:

1. Added a hit counter in the Cache data structure
2. Added a miss counter in the Cache data structure

Function updates:

1. Updated the cache_new function to initialize the hit and miss counter to 0.
2. Updated the cache_access function to update the values for hit and miss tracker in point 1 above.
3. Updated the cache_find_victim function to support the Part F implementation using the DWP flag check
    a. It orders the cache lines for each core in the set based on the last_access_time.
    b. Then computes the utot by iterating through the possible partition values.

c. For each partition is calculates the contribution towards utot for both cores by changing the number of ways allocated to each and using the hits for each way which would have been misses if the allocation was different. This is done going from the MRU to LRU made possible due to the ordering done in point “a” above.
d. For the partition value which leads to the highest utot, it attempts to find the victim through the generic function created for Part E (cache_find_victim_from_partition). This takes an additional input: the partition location.
e. Now, for finding the victim, it checks the data occupied by each core. Now for the current core, if the other core is occupying more lines than its quota, then it finds the LRU from that cores allocation to evict, else it finds the LRU From its own allocation.

Results and Observations:

The implementation is tested on all 3 combinations of mixes between the traces used for Part D and E.

Combination	Lab Part	Cycles	CORE_0_IPC	CORE_1_IPC	L2CACHE_READ_MISS_PERC	L2CACHE_WRITE_MISS_PERC	L2CACHE_TOTAL_MISS_PERC
bzip2 and libq	D	211035456	0.802	0.474	71.641	0.046	71.687
	E	162168181	0.805	0.483	70.846	0.034	70.88
	F	206780346	0.849	0.484	67.288	0.022	67.31
bzip2 and lbm	D	166699051	0.742	0.600	55.476	0.019	55.495
	E	170577421	0.801	0.586	53.434	0.008	53.442
	F	166571431	0.844	0.600	73.711	53.147	126.858
lbm and libq	D	232025356	0.598	0.431	70.265	0.002	70.267
	E	213973426	0.584	0.467	70.265	0.002	70.267
	F	213565186	0.561	0.468	70.265	0.002	70.267

The above table clearly shows that for the first combination of bzip2 and libq, the L2 read and miss percentages (L2CACHE_READ_MISS_PERC + L2CACHE_WRITE_MISS_PERC) are lower for Lab Part F implementation above. **The difference in total percentages for mix1 Part F compared to Part D and E is 4.377% and 3.57% respectively, which is higher than the 1% difference required for full credit in Part F of the lab.**

For the other combinations, the total number of cycles required for the programs are lower for Part F compared to E and F because of the change in data access pattern due to the Utility based partitioning implementation. This is due to combination of the L2 data access pattern and DRAM policy (same for all parts). So, again the Part F implementation is better.

Reason for better performance:

1. Dynamic Way partitioning performs better as:
    a. It considers the type and part of workload running. For example, if the workload of memory access changes between two workloads running in parallel, where workload 1 is running intensive instructions during first 1 million instructions while workload 2

runs more memory intensive memory intensive instruction, irrespective of how the partition is statically set, the workload 1 or 2 performance would suffer during 0-1 million instructions or 1-2 million instructions respectively, depending on which workload is given a bigger chunk in the partition.

   b. It does not take into account, access patterns. For example, if the architecture is such that it has private L1 cache and shared L2 cache. If it is a multi-core running two workloads, with workload 1 running memory intensive commands but all hit in L1 cache, whereas workload 2 is running a less memory intensive commands which hit in the L2 level. So say, effectively the memory accesses and data demand from the L2 cache is more for workload 2. In this case, it would be more beneficial to allocate a larger chunk of the partition to workload 2. But static partition if decided based on number of memory accesses, would skew it towards workload 1, which will negatively impact performance.

2. Utility based Cache Partitioning performs better as it considers the utility of the ways in each set based on their pattern of hits/misses to decide whether the system would benefit (based on change in number of misses) by allocating more ways to either core. In this, way it accounts for the dynamic nature of the workload and calculates the new values based on the prior history of data access at the way level granularity, giving it more control.

Thus, due to the above mentioned reasons, Dynamic Way partitioning and specifically Utility based Cache Partitioning performs better than the other static approaches in Part D and E.